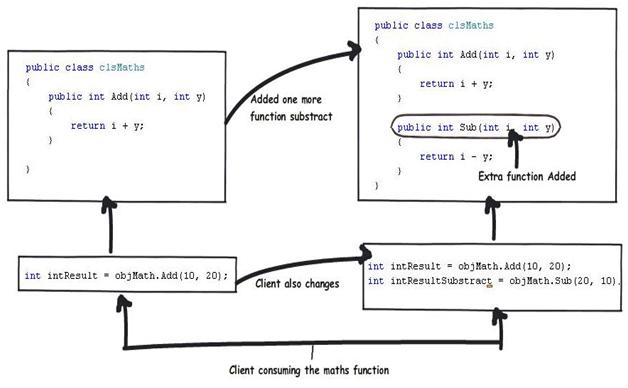
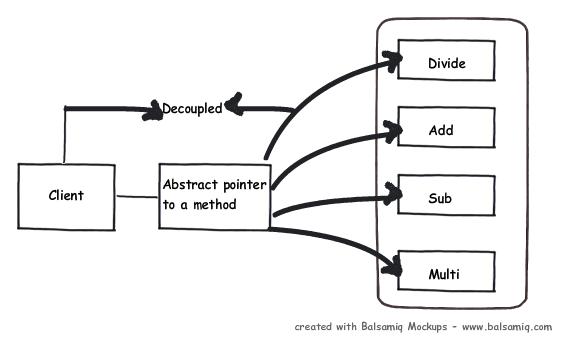
**Abstraction problems of methods and functions**

Before we move ahead and we talk about delegates let’s try to understand what problem does delegate solve. Below is a simple class named ‘ClsMaths’ which has a simple ‘Add’ function. This class ‘ClsMaths’ is consumed by a simple UI client. Now let’s say over a period of time you add subtraction functionality to the ‘ClsMaths’ class, your client need to change accordingly to accommodate the new functionality.  
In other words addition of new functionality in the class leads to recompiling of your UI client.

x`

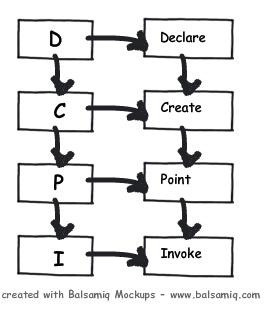
In short the problem is that there is a tight coupling of function names with the UI client. So how can we solve this problem?. Rather than referring to the actual methods in the UI / client if we can refer an abstract pointer which in turn refers to the methods then we can decouple the functions from UI.  
Later any change in the class ‘ClsMath’ will not affect the UI as the changes will be decoupled by the Abstract pointer. This abstract pointer can be defined by using delegates. Delegates define a simple abstract pointer to the function / method.



Ok, now that we have understood the tight coupling problem and also the solution, let’s try to understand how we can define a simple delegate.

**How to declare a delegate?**

To implement a delegate is a four step process declare, create, point and invoke.



The first step is to declare the delegate with the same return type and input parameters. For instance the below function ‘add’ has two input integer parameters and one integer output parameter.

Hide   Copy Code

private int Add(int i,int y)

{

return i + y;

}

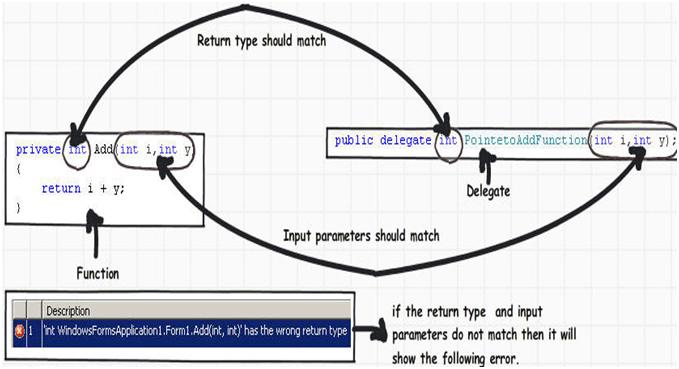
So for the above method the delegate needs to be defined in the follow manner below. Please see the delegate keyword attached with the code.

Hide   Copy Code

// Declare delegate

public delegate int PointetoAddFunction(int i,int y);

The return type and input type of the delegate needs to be compatible, in case they are not compatible it will show the below error as shown in the image below.



The next step (second step) is to create a delegate reference.

Hide   Copy Code

// Create delegate reference

PointetoAddFunction myptr = null;

The third step is to point the delegate reference to the method, currently we need to point the delegate reference to the add function.

Hide   Copy Code

// Point the reference to the add method

myptr = this.Add;

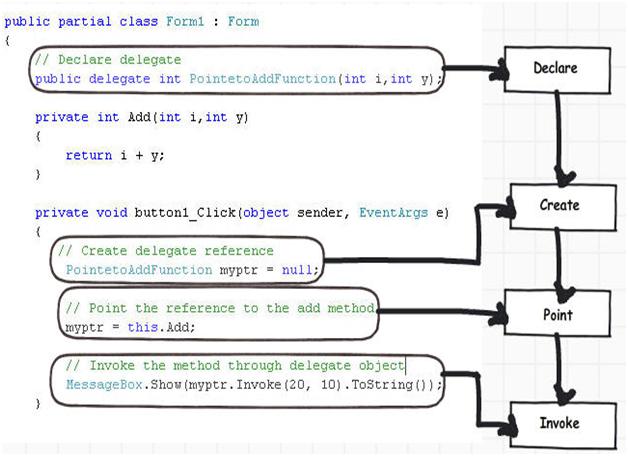
Finally we need to invoke the delegate function using the “Invoke” function of the delegate.

Hide   Copy Code

// Invoke the delegate

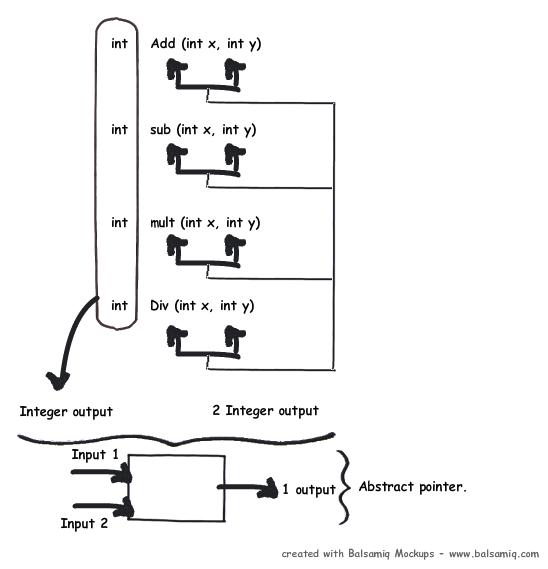
myptr.Invoke(20, 10)

Below figure sums up how the above four step are map with the code snippet.



**Solving the abstract pointer problem using delegates**

In order to decouple the algorithm changes we can expose all the below arithmetic function through an abstract delegate.



So the first step is to add a generic delegate which gives one output and takes two inputs as shown in the below code snippet.

Hide   Copy Code

public class clsMaths

{

public delegate int PointerMaths(int i, int y);

}

The next step is to expose a function which takes in operation and exposes an attached delegate to the UI as shown in the below code snippet.

Hide   Shrink   Copy Code

public class clsMaths

{

public delegate int PointerMaths(int i, int y);

public PointerMaths getPointer(int intoperation)

{

PointerMaths objpointer = null;

if (intoperation == 1)

{

objpointer = Add;

}

else if (intoperation == 2)

{

objpointer = Sub;

}

else if (intoperation == 3)

{

objpointer = Multi;

}

else if (intoperation == 4)

{

objpointer = Div;

}

return objpointer;

}

}

Below is how the complete code snippet looks like. All the algorithm functions i.e. ‘Add’ , ‘Sub’ etc are made private and only one generic abstract delegate pointer is exposed which can be used to invoke these algorithm function.

Hide   Shrink   Copy Code

public class clsMaths

{

public delegate int PointerMaths(int i, int y);

public PointerMaths getPointer(int intoperation)

{

PointerMaths objpointer = null;

if (intoperation == 1)

{

objpointer = Add;

}

else if (intoperation == 2)

{

objpointer = Sub;

}

else if (intoperation == 3)

{

objpointer = Multi;

}

else if (intoperation == 4)

{

objpointer = Div;

}

return objpointer;

}

private int Add(int i, int y)

{

return i + y;

}

private int Sub(int i, int y)

{

return i - y;

}

private int Multi(int i, int y)

{

return i \* y;

}

private int Div(int i, int y)

{

return i / y;

}

}

So at the client side the calls becomes generic without any coupling with the actual method names like ‘Add’ , ‘Sub’ etc.

Hide   Copy Code

int intResult = objMath.getPointer(intOPeration).Invoke(intNumber1,intNumber2);

**Multicast delegates**

In our previous example we have see how we can create a delegate pointer to a function or method. We can also create a delegate which can point to multiple functions and methods. If we invoke such delegate it will invoke all the methods and functions associated with the same one after another sequentially.  
Below is the code snippet which attaches 2 methods i.e. method1 and method2 with delegate ‘delegateptr’. In order to add multiple methods and function we need to use ‘+=’ symbols. Now if we invoke the delegate it will invoke ‘Method1’ first and then ‘Method2’. Invocation happen in the same sequence as the attachment is done.

Hide   Copy Code

// Associate method1

delegateptr += Method1;

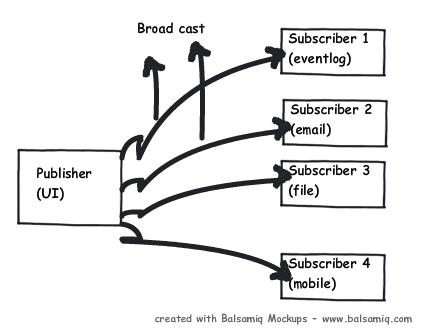
// Associate Method2

delegateptr += Method2;

// Invoke the Method1 and Method2 sequentially

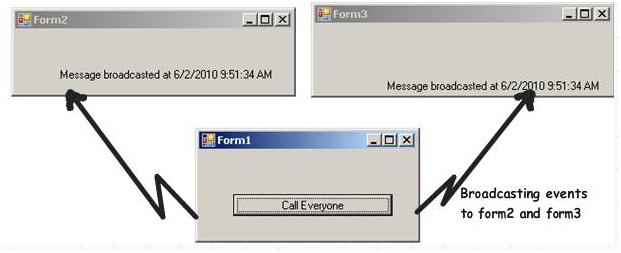
delegateptr.Invoke();

So how we can use multicast delegate in actual projects. Many times we want to create publisher / subscriber kind of model. For instance in an application we can have various error logging routine and as soon as error happens in a application you would like to broadcast the errors to the respective components.



**Simple demonstration of multicast delegates**

In order to understand multicast delegates better let’s do the below demo. In this demo we have ‘Form1’, ‘Form2’ and ‘Form3’. ‘Form1’ has a multicast delegate which will propagate event to ‘Form2’ and ‘Form3’.



At the form level of ‘Form1’ (this form will be propagating events to form2 and form3) we will first define a simple delegate and reference of the delegate as shown in the code snippet below. This delegate will be responsible for broadcasting events to the other forms.

Hide   Copy Code

// Create a simple delegate

public delegate void CallEveryOne();

// Create a reference to the delegate

public CallEveryOne ptrcall=null;

// Create objects of both forms

public Form2 obj= new Form2();

public Form3 obj1= new Form3();

In the form load we invoke the forms and attach ‘CallMe’ method which is present in both the forms in a multicast fashion ( += ).

Hide   Copy Code

private void Form1\_Load(object sender, EventArgs e)

{

// Show both the forms

obj.Show();

obj1.Show();

// Attach the form methods where you will make call back

ptrcall += obj.CallMe;

ptrcall += obj1.CallMe;

}

Finally we can invoke and broadcast method calls to both the forms.

Hide   Copy Code

private void button1\_Click(object sender, EventArgs e)

{

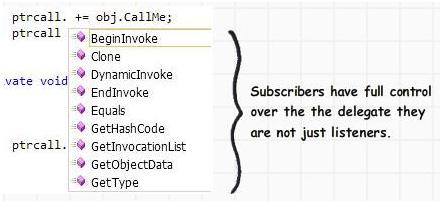
// Invoke the delegate

ptrcall.Invoke();

}

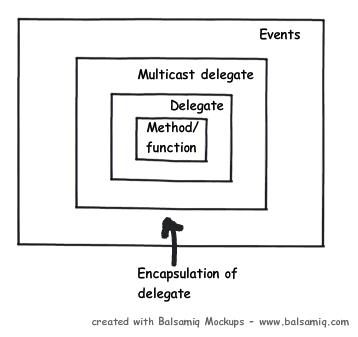
**Problem with multicast delegates – naked exposure**

The first problem with above code is that the subscribers (form2 and form3) do not have the rights to say that they are interested or not interested in the events. It’s all decided by ‘form1’.  
We can go other route i.e. pass the delegate to the subscribers and let them attach their methods if they wish to subscribe to the broadcast sent by ‘form1’. But that leads to a different set of problems i.e. encapsulation violation.  
If we expose the delegate to the subscriber he can invoke delegate, add his own functions etc. In other words the delegate is completely naked to the subscriber.



**Events – Encapsulation on delegates**

Events help to solve the delegate encapsulation problem. Events sit on top of delegates and provide encapsulation so that the destination source can only listen and not have full control of the delegate object.  
Below figure shows how the things look like:-  
• Method and functions are abstracted /encapsulated using delegates  
• Delegates are further extended to provide broadcasting model by using multicast delegate.  
• Multicast delegate are further encapsulated using events.



**Implementing events**

So let’s take the same example which we did using multicast delegates and try to implement the same using events. Event uses delegate internally as event provides higher level of encapsulation over delegates.  
So the first step in the publisher (‘Form1’) we need to define the delegate and the event for the delegate. Below is the code snippet for the same and please do notice the ‘event’ keyword.  
We have defined a delegate ‘CallEveryOne’ and we have specified an event object for the delegate called as ‘EventCallEveryOne’.

Hide   Copy Code

public delegate void CallEveryone();

public event CallEveryone EventCallEveryOne;

From the publisher i.e. ‘Form1’ create ‘Form2’ and ‘Form3’ objects and attach the current ‘Form1’ object so that ‘Form2’ and ‘Form3’ will listen to the events. Once the object is attached raise the events.

Hide   Copy Code

Form2 obj = new Form2();

obj.obj = this;

Form3 obj1 = new Form3();

obj1.obj = this;

obj.Show();

obj1.Show();

EventCallEveryOne();

At the subscriber side i.e. (Form2 and Form3) attach the method to the event listener.

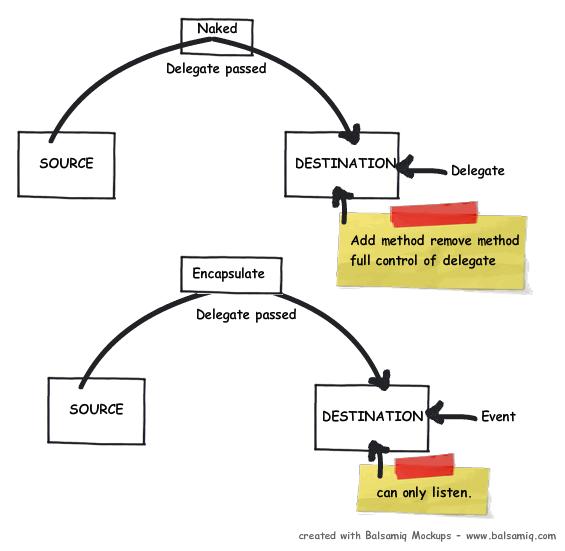
Hide   Copy Code

obj.EventCallEveryOne += Callme;

This code will show the same results as we have got from multicast delegate example.

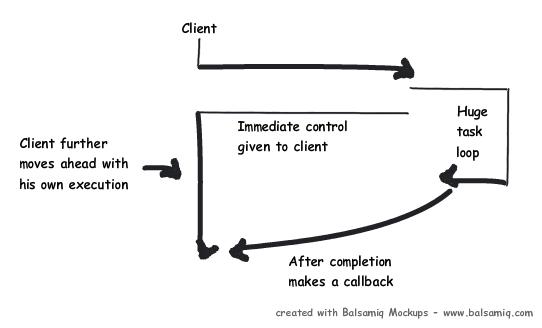
**Difference between delegates and events**

So what’s really the difference between delegates and events other than the sugar coated syntax of events. As already demonstrated previously the main difference is that event provides one more level of encapsulation over delegates.  
So when we pass delegates it’s naked and the destination / subscriber can modify the delegate. When we use events the destination can only listen to it.



**Delegates for Asynchronous method calls**

One of the other uses of delegates is asynchronous method calls. You can call methods and functions pointed by delegate asynchronously.  
Asynchronous calling means the client calls the delegate and the control is returned back immediately for further execution. The delegate runs in parallel to the main caller. When the delegate has finished doing his work he makes a call back to the caller intimating that the function / subroutine has completed executing.



To invoke a delegate asynchronously we need call the ‘begininvoke’ method. In the ‘begininvoke’ method we need to specify the call back method which is ‘CallbackMethod’ currently.

Hide   Copy Code

delegateptr.BeginInvoke(new AsyncCallback(CallbackMethod), delegateptr);

Below is the code snippet for ‘CallbackMethod’. This method will be called once the delegate finishes his task.

Hide   Copy Code

static void CallbackMethod(IAsyncResult result)

{

int returnValue = flusher.EndInvoke(result);

}

**Summarizing Use of delegates**

There are 6 important uses of delegates:-  
**1. Abstract and encapsulate a method (Anonymous invocation)**  
This is the most important use of delegates; it helps us to define an abstract pointer which can point to methods and functions. The same abstract delegate can be later used to point to that type of functions and methods. In the previous section we have shown a simple example of a maths class. Later addition of new algorithm functions does not affect the UI code.  
  
**2. Callback mechanism**Many times we would like to provide a call back mechanism. Delegates can be passed to the destination and destination can use the same delegate pointer to make callbacks.  
  
**3. Asynchronous processing**By using ‘BeginInvoke’ and ‘EndInvoke’ we can call delegates asynchronously. In our previous section we have explained the same in detail.  
  
**4. Multicasting - Sequential processing** Some time we would like to call some methods in a sequential manner which can be done by using multicast delegate. This is already explained in the multicast example shown above.  
  
**5. Events - Publisher subscriber model**We can use events to create a pure publisher / subscriber model.